Introduction to
Parallel and Distributed Computing
Exercise 2 (May 5)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

The result is to be submitted by the deadline stated above via the Moodle interface as a .zip or .tgz file which contains

- a PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - the source code of the original program,
  - the demonstration of a sample solution of the program,
  - the output of the parallelizing compilation of this source with an explanation of this output and the inhibitors of parallelism (if any),
  - the source code of the modified program,
  - the demonstration of a sample solution of the program,
  - the output of the parallelizing compilation of this source with an explanation of this output (comparing it with the previous one),
  - a benchmark of the sequential and of the parallel program in the style of Exercise 1.
- the source (`.c`/`.f`) files of the original program and the modified program.

Exercise 2: Semi-Automatic Parallelization and OpenMP

Gaussian Elimination is a well-known algorithm for solving a linear system $Ax = b$ of some dimension n (i.e. we are given a matrix A of size $n \times n$ and a vector b of length n and we want to find that vector x of length n such that the equation holds).

The algorithm consists of two steps:

1. The system is converted to an upper-triangular system $Tx = e$ (i.e. all coefficients below the main diagonal of T are zero) which has the same solution(s) as the original system.

2. The new system $Tx = e$ is solved by backward substitution (we determine the solution $x_{n-1} = e_{n-1}$, and substitute the solution in the given system which produces a new upper-triangular system of dimension $n-1$ which can be solved in the same way).

Details about the algorithm are given in the file `GaussianElimination.PDF` in the "Restricted Area" of the course site (Figure 9-13 describes triangulation, Figure 9-3 describes back-substitution).

While Gaussian Elimination is typically not used when the coefficients in A and b are floating point numbers (here mainly iterative methods are used for determining approximate solutions), it plays an important role if the coefficients are integer or rational numbers and the equation is to be solved *exactly* (as it is done in computer algebra systems).

Sequential Program

Your first task is to write in C (or Fortran) a function `solve(A, b)` that returns the solution of the linear system determined by matrix A and vector b (or `null`, if there is no solution or there exist multiple solutions).

In your implementation, you may use for coefficients and results simply floating point numbers (C type `float`) but, in order to simulate the larger computation time of arbitrary precision arithmetic, use in your benchmarks the function

```
float smult(float a, float b)
{
  float c = 0;
  for (int i=0; i<10000; i++) { c = 1-c*c; }
  return a*b+c;
}
```

as a "slow multiplication" operation whenever the multiplication of coefficients is required.

You may construct a "straight-forward" version of the algorithm where any non-zero coefficient may serve as a pivot element in the triangulization step (i.e. is not necessary to take the element with the maximum absolute value, as it is done in the algorithm of Figure 9-1).

Demonstrate the correctness of your program by solving a random 4×4 system and giving the output of the program (system and solution).

Benchmark the execution time of `solve` for randomly initialized matrices of dimensions $N = 256$ and $N = 378$ (in case these values take much too long or much too short, you may adapt N correspondingly).

Parallel Program

The more time-consuming part of Gaussian Elimination is the conversion of the system into upper-triangular form where in n iterations one row of the system after the other is converted into the right form. In iteration i of the algorithm, all coefficients of A below and to the right of position (i,i) have to be processed (see Figure 9-12); since this can be done independently for each coefficient, we have basically a parallel algorithm.

So compile your previously constructed program (without any changes to the source code) with the options `-parallel -par-report3` and investigate and explain the compilation report. Most probably, however, the compiler will not have been able to parallelize the program effectively, so explain the inhibitors of parallelization in detail.

For parallelization, you may choose one of the options below (if you choose both, you will get 30% bonus grade). Whatever option you choose, demonstrate the correctness of your parallel program in the same way as for the original one.

Benchmark your program for the values of N given above and for $P = 1,2,4,8,16$ processors; you should get substantial speedups. In your (sequential and parallel) benchmarks, apply the same strategy as explained in Exercise 1 (you may just perform three runs and take the average).

Option A: Semi-Automatic Parallelization

Modify your program in such a way that the automatically parallelizing compiler gives a better result. For instance, rather than using an array *marked* (as is done in Figure 9-1) it might be wise to permute the rows of the matrix such that a contiguous range of rows is processed in the core loop (you have to record the permutation for constructing the coefficients of x in the right order). Also it might be useful to to copy the row with the "picked" pivot value into an auxiliary vector and use this rather than $A[picked]$ in the k-loop. In general, try to make the j/k-loops as simple as possible such that the compiler can make use of the inherent parallelism. Compile the program with the parallelization option switched on and explain the compilation output.

Option B: OpenMP

Modify the program (if necessary) such that the iterations of the j-loop can be (algorithmically) performed independently of each other. Then parallelize the program by annotating the j-loop with OpenMP pragmas such that the loop gets executed in parallel; do not forget to mark "private" variables appropriately. Compile the program with options `-openmp -openmp-report2` and explain the compilation output.